Predicting Probable Prices: Using Python to Estimate Stock Market Movements

Hunter Pharr

North Carolina Wesleyan College

Honors Thesis

Contents

Abstract

Stock markets are very difficult to predict, however, with the power of machine learning, the possible movements of a price of stock can be estimated. Two models can be used to assist with this. One is the Monte Carlo simulation which helps account for the randomness that can occur with the stock market. Another less random way to predict stock market movements is with machine learning, specifically long short-term memory. While neither of these models are ever entirely accurate and may not find use among day traders, they can be close with their predictions and be of use to business analysts and budget forecasters.

Predicting the price of a specific stock is historically a daunting process due to constant changes in price that are influenced by many variables such as a company's earning power, inflation, market demand, and current events. The volatility of the stock market has also been a roadblock for economic and financial time series forecasting. While predicting a specific price point of a stock accurately is extremely difficult, predicting the movement of stock prices is an achievable goal. While not all variables that influence stock prices can be accounted for, programming languages such as python can create powerful models to predict future movements in the stock market.

The first objective of this work is to provide an overview of python, the stock market, and time series analyses. The second objective is to showcase how Python, through the power of predictive simulations and recurrent neural networks, can forecast the movement of stock prices. Using simulations such as the Monte Carlo simulation, and more advanced algorithms such as Long Short-term Memory (LSTM) models, python can be a powerful tool to aid in predicting movements in the stock market.

Review of Literature

Introduction to Python

Python is a general-purpose, open-source programming language created by Guido von Rossum in 1990. Python is an open-source program, meaning the original source code is made freely available to users and can be modified. Due to its open-source nature, Python evolved into an extremely versatile programming language.

Python was not a commonly used language for the first years of its development. However, with the arrival of Python 2.0 in 2000, multiple improvements were made to the

language. The arrival of Python 2.0 allowed users to create libraries that were seamlessly interfaced into Python's language (Guttag, 2016). Since users of Python have free access to its source code, the continuing development of Python became a community-based activity.

Python 3.0 released in early 2008 and cleaned up a lot of the inconsistencies in Python's design. While Python 3 did aid in cleaning up a lot of the issues with Python 2, Python 3 was not backwards compatible with Python 2, therefore all programs written for Python 2 could not be run on Python 3.

In 2008 Python introduced a package manager known as pip which made the installation process of libraries and other packages in Python much easier. These libraries are important for advanced scripts in python as they can import more advanced functions that are not included in the standard versions of python. Some of the most popular libraries that are used with python are NumPy, pandas, and matplotlib. For details on what these libraries include, see appendix A.

Stock Market Forecasting

Stock market prices are historically influenced by a variety of micro and macroeconomic variables. Most variables that can influence the price of a stock are usually based on the performance of other industries or the market. These include rate of inflation, the price of various metals, oil prices, Gross Domestic Product (GDP), employment rate, and interest rates (Jain, Gupta, Singh, 2018). Non-market variables can include government policies, and natural or man-made disasters (Jain et al., 2018). It is difficult to determine which of these variables influence stock prices the most, however, all these factors work in tandem to add to the perceived randomness of the stock market.

While the many variables that impact the price of stock can make stock prices seem chaotic, there is a pattern of trends that the stock market goes through. According to Jeffery and Yale Hirsch of *The Stock Trader's Almanac,* "while stocks do indeed fluctuate, they do so in well-defined, often predictable patterns. These patterns recur too frequently to be a result of chance or coincidence" (Balenthiran, 2013). This pattern that the stock market goes through is known as the business or economic cycle. A business cycle usually contains periods of growth, stagnation and decline (Balenthiran, 2013). Once a cycle ends, a new period of growth emerges, and the cycle continues. Since the stock market moves in a cycle, long-term forecasting of the stock market can be possible with predictive models.

Predictive Modeling

According to Merriam-Webster, the definition of a model, in the context of using numbers, is a system of data and inferences presented as a mathematical description of an entity or state of affairs. Models help with visualizing data to decipher an incoherent mass of numbers. To add to the definition of modeling, a prediction, regarding data science, means to estimate an unknown value.

In data science, a predictive model is a formula for estimating the unknown value of interest, known as the target (Provost, Fawcett, 2014). This formula can be a mathematical equation, a statement, or both. This is different from descriptive modeling as the primary purpose of descriptive modeling is to gain insight into a phenomenon or process. A descriptive model is judged on its intelligibility while a predictive model is judged on its performance in forecasting (Provost et al., 2014). While the definitions of predictive and descriptive models seem strict, some of the techniques these models use can be interchangeable, and one model can serve as both a predictive and descriptive model.

Predictive models can be divided into two types. The first type of predictive model is known as a classification model. These models usually involve binary outputs in the form of 0 or 1. These binary outputs can be changed to be more descriptive such as determining whether a specific stock price will increase or decrease; the binary outputs are increase and decrease. The second type of predictive model is a regression model. Regression models predict a specific number. An example would include predicting the exact price that a stock may achieve over a specified period. The primary focus of this study will be on regression models.

While predictive models are designed for looking at the future, these models need historical data to properly judge trends in a dataset. A dataset is a table that contains a group of examples or instances, which can be referred to as data points. Predictive models become more accurate if a dataset has more data points. Long-term models tend to be more accurate than shorter-term models since long-term models can contain more data points (Moody, 2013). Developments in technology have allowed larger datasets to be available which can result in more accurate models.

The concept of predictive analysis can date as far back as the late 17th century, however, recent technological advances have brought predictive analysis to the forefront of business strategies (Predictive Success Corporation, 2019). Easier to use software is becoming more prevalent which allows predictive analytics to be accessible not only to mathematicians and statisticians, but business analysts as well (SAS, Predictive Analytics). With the internet, the volumes, types and availability of data are growing, which is providing more information for datasets, allowing analysts to make more accurate models (SAS). Computer hardware is also becoming faster and cheaper. The convergence of easier software, more data, and better hardware has allowed data analysts to use complex predictive algorithms with greater ease.

Monte Carlo Simulation

Monte Carlo simulations (MCS) are used to simulate, or imitate, real systems and account for the randomness and uncertainty of variables in these systems (Krizanova, Masarova, Majercak, Buc, 2013). The MCS' name is borrowed from the casinos in Monte Carlo, Monaco. MCS relies on random draws from a probability function (Krizanova et al., 2013). MCS are used in a variety of fields such as finance, project management, insurance and manufacturing.

In an ideal setting, for more accurate results, it is recommended to use the MCS method in completely random scenarios. MCS is used when there are multiple factors that can influence the performance of projects. In this study, the MCS will be used to analyze possible movements in the price of a specific company's stock. While the stock market is not a random set of numbers, it does have many variables that influence the outcome of closing prices.

The primary objective of using an MCS is to generate a large quantity of scenarios and values for each scenario (Krizanova et al., 2013). For example, an MCS can be given historical data on a company's stock price and provide several random scenarios predicting how a stock price will trend. The MCS can also be used to in sensitivity analysis by showing the possible changes of a specified criteria and showing the potential changes in the values of risk factors that can affect the criteria (Krizanova et al., 2013). An MCS can display these the results of these scenarios in numerical or graphical form. Since the MCS will be used for stock market prices, the outputs will be shown in graphical form.

The MCS can also be used in a variety of probability distributions. These distributions include, but are not limited to, normal, lognormal and uniform (Palisade). While there are multiple distributions that the MCS can use, the distribution that the MCS will use in this paper

is the lognormal distribution. Lognormal distributions are used when the values in a dataset are positively skewed. Additionally, the values in lognormal distributions do not drop below zero and have unlimited positive potential (Palisade). This is representative of the stock market since stock prices never drop below zero and, in a historical view, stock prices normally trend positively.

With advances in computational power, Monte Carlo simulations are becoming easier to use for data analysts. MCS usually involves a multitude of mathematical calculations which a computer can perform in seconds. With advancements in computational power, MCS can be created in many different applications. While this paper will demonstrate MCS in Python, MCS can also be created in other applications such as Microsoft Excel. While MCS can be useful for observing stock price trends, the rise in computational ability has led to more advanced ways of analyzing data trends.

Machine Learning

Machine learning is the science of programming computers to learn from data (Geron, 2018). While conversations about machine learning and artificial intelligence (AI) tend to overlap, these two are not the same. Machine learning is the idea that machines should be able to learn and adapt through experience, while AI refers to a broader idea where topics such as machine learning, deep learning, and other methods can solve problems (SAS, Artificial Intelligence). Machine learning is usually a singular algorithm designed to solve a problem, while AI is a collection of various machine learning algorithms designed to solve and analyze an issue.

One of the first instances of a machine learning algorithm is the spam filter. Spam filters are usually trained on a set of data provided by users. Users will flag email as spam and the spam filter will observe the email and look for similarities between the currently flagged email and previously flagged emails (Muller, Guido, 2018). The spam filter "learns" from its dataset and uses this knowledge to find spam emails.

Machine learning algorithms are more adaptable than traditional computer programs. Some programs that require a long list of rules, or if and else statements can be simplified into a machine learning algorithm. A spam filter program could be created that states all the common words and phrases used in spam emails using if, else statements, however this is not very efficient (Geron, 2018). A machine learning algorithm automates this process by comparing previously flagged data against current data and provide an output. Machine learning can also handle large amounts of data or data that fluctuates rapidly, which can be useful for observing the stock market.

Types of Machine Learning

Machine learning can be divided into different types that classify how the algorithm is given data, supervised or unsupervised, how the algorithm is receiving its training data, batch or online learning, and if the algorithm operates by comparing data points or finding patterns, instance or model based learning. With supervised and unsupervised learning, if training data is labeled in the desired outputs, it is supervised learning, if the data is not labeled, it is unsupervised (Geron, 2018). With batch learning, all the data is provided at one time, with online learning the data is fed to the algorithm incrementally (Geron, 2018). Instance based learning generally compares previous examples of data it has received to come up with a singular result. With model-based learning, an algorithm uses the provided data and attempts to find patterns or

make predictions with this data through creating models (Geron, 2018). While these learning styles may seem unrelated, multiple learning styles can be used with one algorithm. For example, a price prediction algorithm could use unsupervised, batch and model-based learning to provide its results.

Neural Networks

A more complex version of machine learning is known as neural networks. Neural networks are modeled loosely after the human brain and consist of thousands or millions of processing nodes that are interconnected (Hardesty, 2017). Neural networks take their training data and assign a number, or weight, to them. These weights start as random numbers when the network begins training. When a neural network receives a data point it multiplies the assigned weight to the given data point. It then adds the resulting numbers together and if the number is above a specified point, it passes the data through the node, if the number is below the target, the node passes no data. (Hardesty, 2017). During the training process, the weights and specified points are adjusted until training data with similar labels begin to show similar outputs (Hardesty, 2017).

Recurrent Neural Networks (RNN)

A recurrent neural network (RNN) is a specific type of neural network designed for predictions. RNNs are named recurrent because the same calculation and tasks are performed for every element in a sequence. The objective of RNNs is to predict the next step in a sequence of observations when observing previous data in a sequence (Namin, Namin, 2018). With RNNs, earlier stages of data need to be remembered in order to forecast future events, therefore, the

layers in a RNN act as internal storage for the information gathered in the earlier stages of reading data.

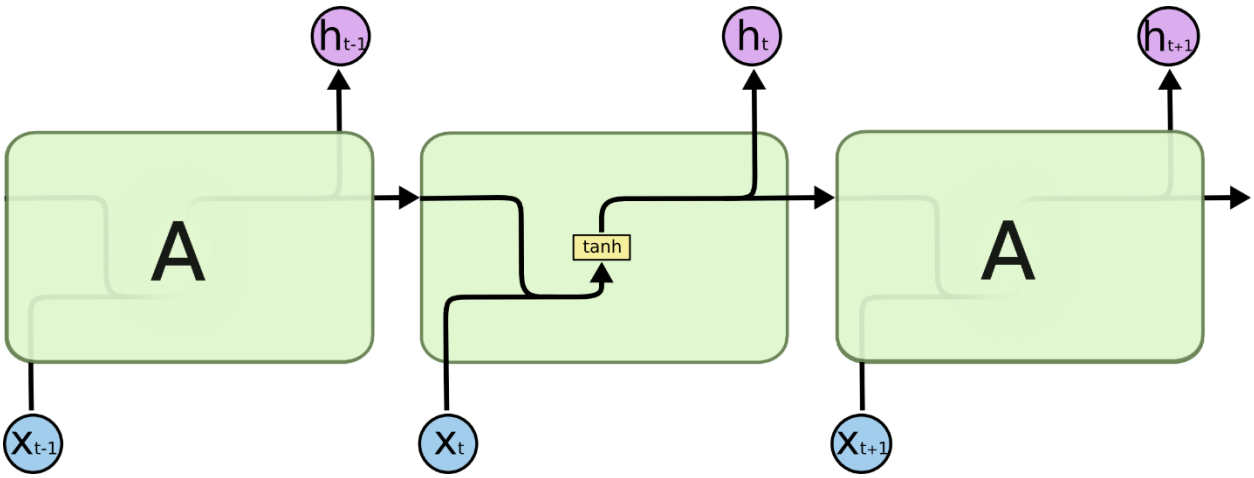

Figure 1 – RNN Module (Olah, 2015)

Figure 1 is an example of the basic structure of an RNN. All RNNs have a chain of repeating structures of neural networks. These structures are demonstrated inside the green rectangles. The arrows in Figure 1 demonstrate how data flows through these structures and is modified. RNNs are simple in their default structure since they only have one layer that interacts with data. In Figure 1, this RNN has only one tanh layer that modifies the output.

While there are many benefits for predictive modeling using RNNs, there are also some drawbacks. A basic problem with RNNs is the computational requirements. If an RNN is required to perform multiple steps of calculations it can become a burden on computer hardware, however, with technological advancements, this is becoming less of an issue.

RNNs deal with some calculation related constraints which are known as the exploding gradient or the vanishing gradient. The exploding and vanishing gradients are commonly referred to as error gradients. “An error gradient is the direction and magnitude calculated during the training of a neural network which is used to update the network weights in the right direction

and by the right amount" (Brownlee, 2019). If the weights that an RNN uses are constantly being updated, this can lead to instability. In extreme cases the value of the weights can become large enough to overflow the network and result in not a number (NaN) values (Brownlee, 2019). Another challenge with RNNs is that these neural networks only remember a small number of sequences and are not well suited to longer sequences of data (Namin et al., 2018). However, both the vanishing gradient and the memory sequence issues can be solved by using long short-term memory.

Long Short-Term Memory (LSTM)

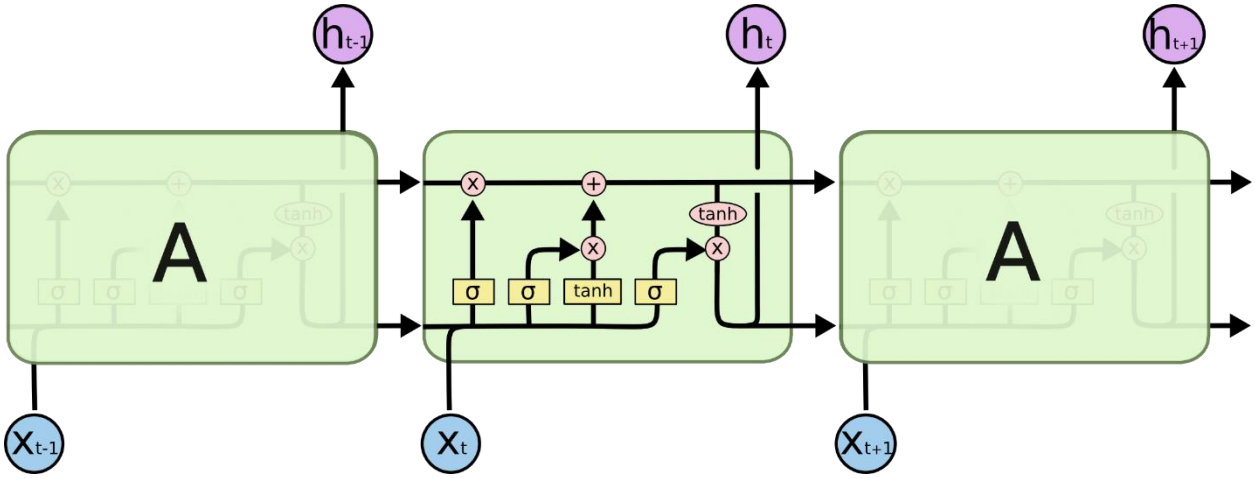

Figure 2 – LSTM Module (Olah. 2015)

LSTM is a type of RNN that has additional features and is designed to memorize the sequence of data it is provided. The cells in an LSTM resemble a transport line that flows from one module to another (Olah, 2015). The purpose of this line is to convey data from past modules and gather them for the present calculations.

LSTM uses what are known as "gates" during data processing. Unlike normal RNNs which use only one layer, LSTM uses 4 layers during its data processing, which are separated into 3 gates. These gates allow data to be disposed, filtered, or added to the next cells (Namin et al., 2018). These gates are identified as the forget gate, memory gate, and output gate.

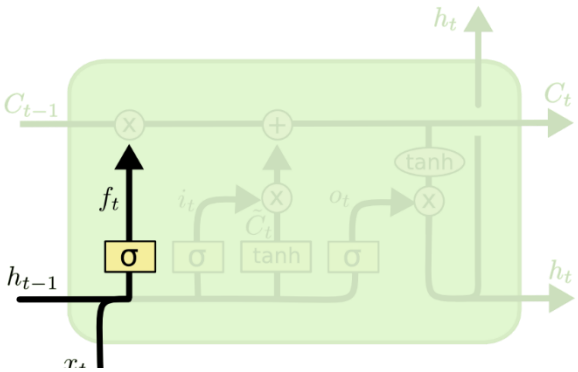

Figure 3 – Forget Gate (Olah, 2015)

The first step in LSTM is to decide what information will be kept in the current cell state. This action is performed by the forget gate. The forget gate outputs a number between zero and one where one means to completely keep the data and zero implies completely ignore the data.

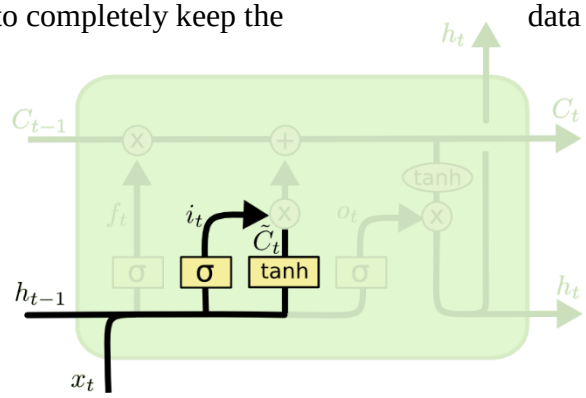

Figure 4 – Input Gate (Olah, 2015)

The next step in LSTM is to decide what information will be stored in the current cell state. In this step the first layer of the input gate decides which values will be updated (i_t). The second layer creates new candidate values that can be added to the cell state (C_t).

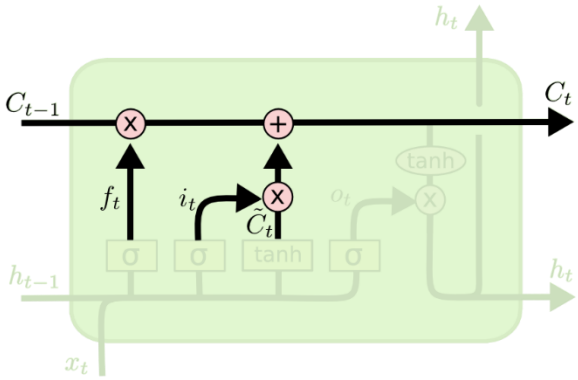

Figure 5 – Update Step (Olah, 2015)

After the input gate makes its calculations, the LSTM updates the old cell state, C_{t-1}, into the new cell state C_t. In this step the old cell state is multiplied by the output from the forget gate (f_t). The output of the input gate ($i_t * C_t$) is then added to the product of the old cell state and the forget gate.

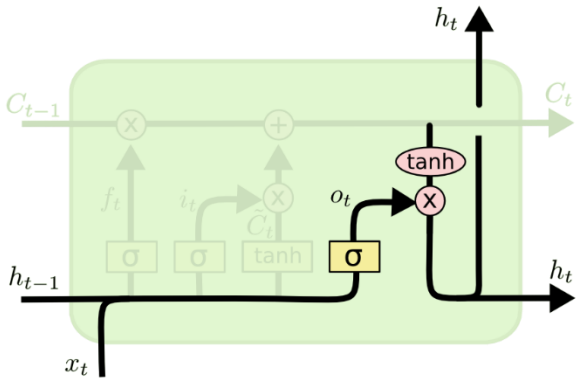

Figure 6 – Output Step

The last step is to determine what the output of the entire cell will be. First a sigmoid layer decides which parts of the cell state will be the output (o_t). Then the cell state will be put

through tanh to push the values between -1 and 1. After the tanh layer is run it will be multiplied by the output from the sigmoid layer. The following is the output equation where h_t represents the final output ($h_t = o_t * \tanh(C_t)$).

While LSTMs perform quite a few complex calculations, these calculations are not required to be coded into python itself. When the LSTM function is called up, the specific python library responsible for LSTM will have a set function it will use to run these equations. However, it is still important for a machine learning programmer to have a general idea of what calculations the algorithm performs.

Program Demonstrations

The following section contains examples of the monte carlo simulation and LSTM algorithm and how they are implemented in python. For these two demonstrations the Google Collaboratory environment is used instead of the traditional python installation. Google Collaboratory is a cloud-based python solution that allows users to create python programs without needing to make any library installations on the user's system. Programs created in Collaboratory are executed on an offsite computer which allows any complex program to be created and run on virtually any machine with access to the internet.

Monte Carlo Demonstration

```
import numpy as np
import pandas as pd
from pandas_datareader import data as wb
import matplotlib.pyplot as plt
from scipy.stats import norm
%matplotlib inline
```

Figure 7 – Monte Carlo Simulation Libraries

Figure 7 shows the different python libraries that will be used in this simulation. For more information on what these libraries provide, see Appendix A.

```
ticker = 'AMD'
data = pd.DataFrame()
data[ticker] = wb.DataReader(ticker, data_source='yahoo', start = '2010-1-1')['Adj Close']
```

Figure 8 – Retrieving Stock Data

The code in Figure 8 shows what stock data will be used and where the data is being obtained. For this example, the stock data for Advanced Micro Dynamics (AMD) will be used. The stock data that is obtained will be stored in a dataframe that allows the data to be easily manipulated. The stock data is being obtained from yahoo.com and uses the adjusted close beginning January 1st, 2010 and ending the date this simulation is executed.

Figure 9 – Log returns

The next step for this monte carlo simulation is to find the logarithmic returns for the stock data. To obtain the logarithmic returns the log of one plus the percent change in the stock data is found. To show some of the results, the last five data points are shown using the .tail function.

```
u = log_returns.mean()
u
```

```
AMD    0.000609
dtype: float64
```

```
var = log_returns.var()
var
```

```
AMD    0.0013
dtype: float64
```

Figure 10 – Mean and Variance

```
drift = u - (0.5 * var)
drift
```

```
AMD   -0.000041
dtype: float64
```

```
stdev = log_returns.std()
stdev
```

```
AMD    0.036061
dtype: float64
```

Next the mean and variance of the log returns are found using the .mean and .var functions. The variance is a measure of how far the stock data is spread out from its average. The variance and mean will be used to find the drift of the data.

Figure 11 – Drift and standard deviation

Figure 11 shows the drift and standard deviation of the stock data. The drift is found by taking the mean of the data and subtracting it from half of the variance. The standard deviation of the stock data is found by using the .std function.

```
t_intervals = 365
iterations = 10
```

Figure 12 – Intervals and Iterations

The next step is setting the amount of days and how many times the simulation will be run. For this example, the program will run for 365 days (intervals) and perform the simulation ten times (iterations).

```
daily_returns = np.exp(drift.values + stdev.values * norm.ppf(np.random.rand(t_intervals, iterations)))
daily_returns
```

```
array([[0.99647752, 0.94788435, 1.02338449, ..., 1.01037061, 1.04169726,
        0.95755786],
       [1.05458817, 1.04328895, 1.0634045 , ..., 0.9227103 , 1.0734544 ,
        0.98042154],
       [0.95715201, 0.98594929, 0.98494457, ..., 0.9630243 , 1.0273351 ,
        0.99481626],
       ...,
       [1.07229929, 0.97798452, 1.04310819, ..., 0.9138777 , 1.02206753,
        1.00209926],
       [1.04070995, 0.98152107, 1.02245107, ..., 1.04543649, 0.96935095,
        1.07618203],
       [1.02825977, 1.01721386, 0.99534022, ..., 0.93952747, 0.99402714,
        0.98336695]])
```

Figure 13 – Daily returns

After finding the drift and standard deviation, and setting the intervals and iterations, these variables are used to find the daily return values. The .exp function uses a mathematical constant known as Euler's number which is approximately 2.718281. In the daily returns function, Euler's number is being raised to the data inside of the parentheses. This can be represented with the function e^x where e is Euler's number and x is the data contained inside the parentheses in figure 13.

The function occurring inside the parentheses in figure 13 represents the process of obtaining the random numbers needed for the monte carlo simulation. The values of the drift and standard deviation are added together and then multiplied by random numbers that are run through the percent point function (.ppf). The percent point function is used to determine the probability that a variable is less than or equal to x where x is a given variable (NIST). The random.rand function is used to create an amount of random numbers specified by the intervals and iterations.

The output of the daily_returns variable in figure 13 is an array that contains all numbers associated with the monte carlo simulation. While not all data points are shown, there are 10 different simulations that each contain 365 data points in this output.

Figure 14 – List Creation

```
S0 = data.iloc[-1]
S0
```

```
AMD    46.580002
Name: 2020-03-27 00:00:00, dtype: float64
```

```
price_list = np.zeros_like(daily_returns)
price_list
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
price_list[0] = S0
price_list
```

```
array([[46.58000183, 46.58000183, 46.58000183, ..., 46.58000183,
        46.58000183, 46.58000183],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ]])
```

After the random numbers are created, the next step is to create a list of zeros for the random numbers to be appended to. The data.iloc function is used to set the variable S0 to the last number in the random array. The iloc function in python sets an index number to the values in an array. The last number in the array can be indexed as -1.

After the S0 variable is set, a list of zeros is created that contains an amount of zeros equal to the amount of random numbers in the daily_returns array. This list of zeros is set using the variable price_list. Once the price_list is created, the S0 variable is called to append the last number of the daily_returns array to the price_list. As shown in Figure 14, each iteration in the price_list array now begins with the number 46.58000183.

```
for t in range(1, t_intervals):
  price_list[t] = price_list[t - 1] * daily_returns[t]
```

```
price_list
```

```
array([[46.58000183, 46.58000183, 46.58000183, ..., 46.58000183,
        46.58000183, 46.58000183],
       [49.12271893, 48.59640137, 49.53338349, ..., 42.97984732,
        50.0015081 , 45.66803721],
       [47.01790893, 47.91358736, 48.78763704, ..., 41.39063755,
        51.36830449, 45.43130576],
       ...,
       [26.75265035, 57.33242762, 39.54648057, ..., 88.19801556,
        85.28570217, 86.49802179],
       [27.8417493 , 56.2729855 , 40.43434132, ..., 92.20542421,
        82.67177631, 93.08761667],
       [28.62855065, 57.24166098, 40.24592605, ..., 86.62952904,
        82.1779893 , 91.53928526]])
```

Figure 15 – Price list completion

After the list of zeros is created, a for loop is needed to bring the rest of the values into the price_list array. In this loop the program is going through each point on the price_list (price_list[t – 1]) and appending the daily returns values to the proper index on the price_list. The output is an array with the proper stock values.

```
plt.figure(figsize = (10, 6))
plt.plot(price_list)
plt.title('Monte Carlo Simulation (AMD)')
plt.xlabel('Num. of Days')
plt.ylabel('Price USD ($)')
```

```
Text(0, 0.5, 'Price USD ($)')
```

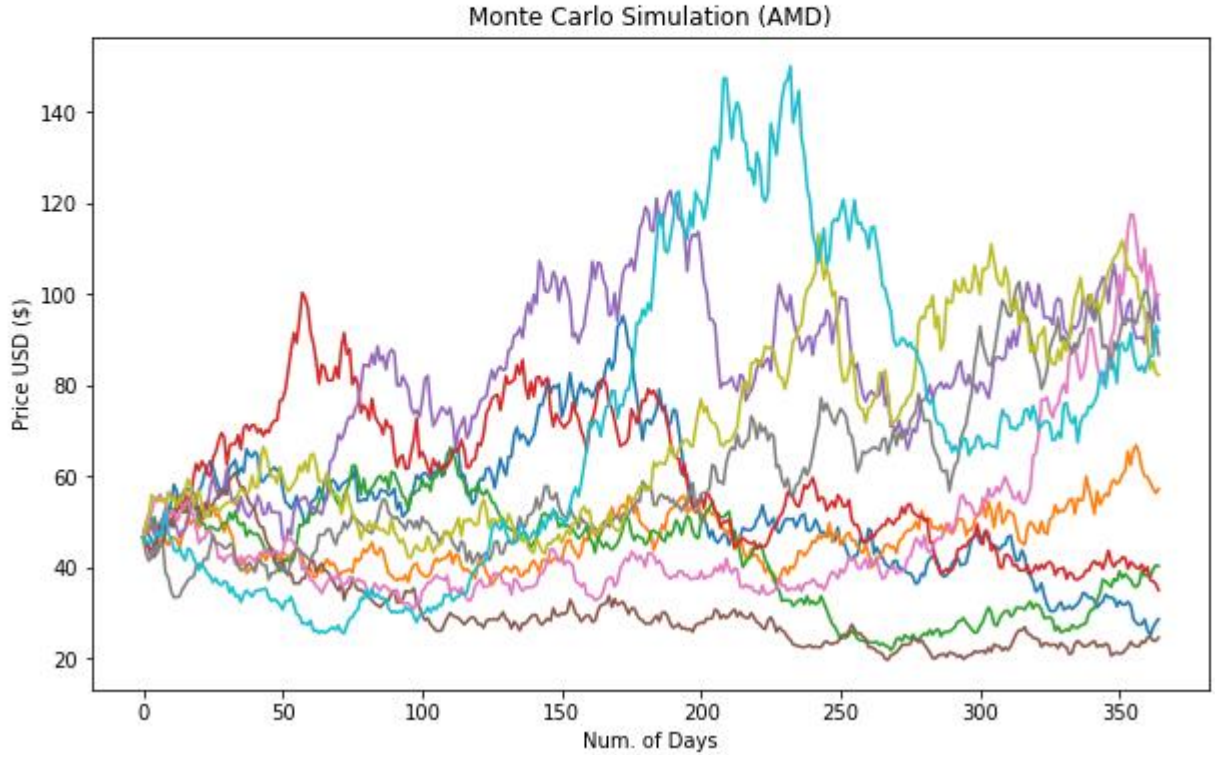

Figure 16 – Plotting the data

Figure 16 shows how this monte carlo simulation looks when the data is plotted. On this graph there are 10 different simulations that each predict 365 days of data. All 10 simulations have the same starting point due to the earlier code, shown in figure 14, where all 10 simulations had the same starting data appended to them.

While the output of this simulation can appear rather erratic, there are some interesting observations on how the simulation predicted the stock price to behave. For example, in Figure 16, 5 different simulations end at the same price range of eighty to 100 dollars, while 4 of the

simulations end at the price range of twenty to forty dollars. The simulation identified by the light blue line drops below the initial value of around forty-eight dollars, spikes to the highest dollar amount of around 150 dollars and decreases to the eighty-dollar price range. The iteration identified by the orange line has the steadiest price as it does not change much from its initial forty-eight-dollar price range; however, when looking at this graph, the orange iteration may be considered an outlier.

This simulation could be useful for analysts at AMD for making a prediction about the general movements that their stock price could take. By looking at the output of this simulation, a business analyst could come to the conclusion that there is a 50 percent chance the AMD stock price increases to the range of eighty to one-hundred dollars, a forty percent chance that the AMD stock price decreases to the thirty to forty dollar range and a ten percent chance that their stock price stays the same over the next 365 days.

LSTM Demonstration

This LSTM demonstration will use historical closing data from AMD to make its own predictions on how a stock price will behave over a specified period. The code for this model is based on a similar model created by randerson112358 which can be viewed at github.com.

```
import math
import pandas_datareader as web
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

Figure 17 – LSTM Libraries

Figure 17 shows the python libraries that will be used for this LSTM algorithm. For more information on these libraries, see appendix A.

```
df = web.DataReader('AMD', data_source='yahoo', start='2010-01-01', end='2020-3-27')
df
```

Date	High	Low	Open	Close	Volume	Adj Close
2010-01-04	9.900000	9.680000	9.790000	9.700000	18748700	9.700000
2010-01-05	9.900000	9.680000	9.710000	9.710000	22145700	9.710000
2010-01-06	9.760000	9.550000	9.680000	9.570000	18643400	9.570000
2010-01-07	9.550000	9.180000	9.510000	9.470000	26806800	9.470000
2010-01-08	9.470000	9.290000	9.370000	9.430000	13752800	9.430000
...	...	...	...	...	...	...
2020-03-23	42.320000	38.950001	40.619999	41.639999	101704700	41.639999
2020-03-24	46.810001	43.990002	44.040001	46.220001	106794200	46.220001
2020-03-25	47.880001	44.430000	46.790001	44.630001	93760400	44.630001
2020-03-26	47.500000	45.400002	45.779999	47.500000	73680200	47.500000
2020-03-27	47.980000	45.900002	46.320000	46.580002	74599200	46.580002

2576 rows × 6 columns

Figure 18 – Stock Data

```
data = df.filter(['Close'])
dataset = data.values
training_data_len = math.ceil( len(dataset) * .8 )
```

Figure 19 – Data Filtering

This LSTM algorithm will be using AMD stock data beginning January 1st, 2010 and ending March 27th, 2020. The stock data is being sourced from yahoo.com.

After the stock data is obtained, the data must be filtered so only the relevant information is used. Since this model will be predicting the closing price, only historical closing prices are

needed. After the data is filtered, the number of rows that the model will be trained on must be set. For this model, only eighty percent of the data will be used to train the model. The math.ceil function is used to round up the resulting number.

```
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)

scaled_data
```

```
array([[0.14106145],
       [0.14123603],
       [0.13879189],
       ...,
       [0.7508729 ],
       [0.80097763],
       [0.78491621]])
```

Figure 20 – Scaling the data

Before data is given to the model to the train, it is generally good practice to scale down the data. Scaling down the data tends to aid neural networks in performing the calculations more efficiently. For this model, the values will be scaled down to be between zero and one using the MinMaxScaler. This scale is then applied to the dataset and saved under the array scaled_data.

```
train_data = scaled_data[0:training_data_len , :]

x_train = []
y_train = []

for i in range(60, len(train_data)):
  x_train.append(train_data[i-60:i, 0])
  y_train.append(train_data[i,0])
  if i<= 61:
    print(x_train)
    print(y_train)
    print()
```

Figure 21 – Creating the training data

After the data is scaled down the next step is to obtain the training data and separate the data into lists. First the data from the scaled_data array is obtained starting from the first datapoint in the array (0) up to the specified length that was set earlier (training_data_len). The resulting output will be saved as train_data.

After the training data is created, two empty lists are created for the training data to be appended to. The x_train lists will contain the independent variables and the y_train list will contain the dependent variables.

To append the train_data to the x_train and y_train lists, a for loop is needed. This model will be using the first 60 data points in the train_data array for its variables. The x_train will have the first sixty datapoints. The y_train list will contain the sixty-first datapoint.

```
x_train, y_train = np.array(x_train), np.array(y_train)

x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
x_train.shape

(2001, 60, 1)
```

Figure 22 – Converting and reshaping the data

After the train lists are appended with the proper data, the lists need to be converted to NumPy arrays. LSTM models need to have data provided in the form of arrays instead of lists.

After the lists are converted to arrays, the train data needs to be reshaped to be three dimensional. Currently the data is two dimensional, however LSTM models need to have three-dimensional data. The dimensions that the LSTM model needs are the number of samples, number of timesteps, and number of features. Samples are the rows in the data, timesteps are past observations and features are the columns in the data (Brownlee, 2019). The data is reshaped using the .reshape function on the x_train list. After the data is reshaped the number of samples

```
model = Sequential()
model.add(LSTM(50, return_sequences =True, input_shape = (x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences = False))
model.add(Dense(25))
model.add(Dense(1))
```

Figure 23 – Creating the LSTM model

is 2001, the number of timesteps is sixty and the number of features is one.

After the training data is properly converted it is time to build the LSTM model. This LSTM model will have two layers. The first layer will have fifty neurons and have an input shape equal to the x_train. The second layer will use fifty neurons as well but has a return sequence of false since only two LSTM layers will be used in this model. After the LSTM layers are created two “dense” layers are created with twenty-five neurons and one neuron respectively.

These dense layers are used for outputting predictions (Brownlee, 2019). The result is a model

```
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Figure 24 – Compiling the model

that contains 4 layers; two LSTM layers and two dense layers.

After the model is created it needs to be compiled. The purpose of compiling the model is to improve the efficiency. This process transforms these layers into efficient series of matrixes that are formatted for the CPU or GPU of the computer (Brownlee, 2019). The optimizer in the compilation step helps improve how the loss function operates while the loss function is a measure of how well the model performed during its training. For this model the adam optimizer and the mean squared error loss functions are used.

```
model.fit(x_train, y_train, batch_size=1, epochs=1)

2001/2001 [==============================] - 125s 63ms/step - loss: 1.7724e-04
```

Figure 25 – Training the model

Once the model is compiled it is ready to train. For this training simulation the batch size of the data and number of epochs are set to one. The batch size is a measure of how many different input and output pairs are created during the training (Brownlee, 2019). An epoch is the amount of times that the model runs through the simulation.

```
test_data = scaled_data[training_data_len - 60:, :]
x_test = []
y_test = dataset[training_data_len:, :]
for i in range(60, len(test_data)):
  x_test.append(test_data[i-60:i, 0])
```

```
x_test = np.array(x_test)
```

```
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
```

Figure 26 – Creating the test dataset

```
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)
```

```
rmse = np.sqrt(np.mean(((predictions - y_test)**2)))
rmse
```

```
2.883582680884054
```

Figure 27 – Making the predictions

Once the model has finished training, the next step is to take the results from this training to make predictions. The process of creating the test_data array is the same as the process for creating the train_data array in figures 21 and 22.

After the test_data array is created the model can begin making its predictions. The purpose of inverse transforming the predictions is to un-scale the values, so they read as normal stock prices.

After the predictions are inverse transformed, the root mean squared error is run to check how accurate the predictions were. Typically, the desired output from the root mean squared

error is to be as close to zero as possible. If the root mean squared error returns zero, then the predictions were exact. The root mean squared error returned a value of 2.88 for this model.

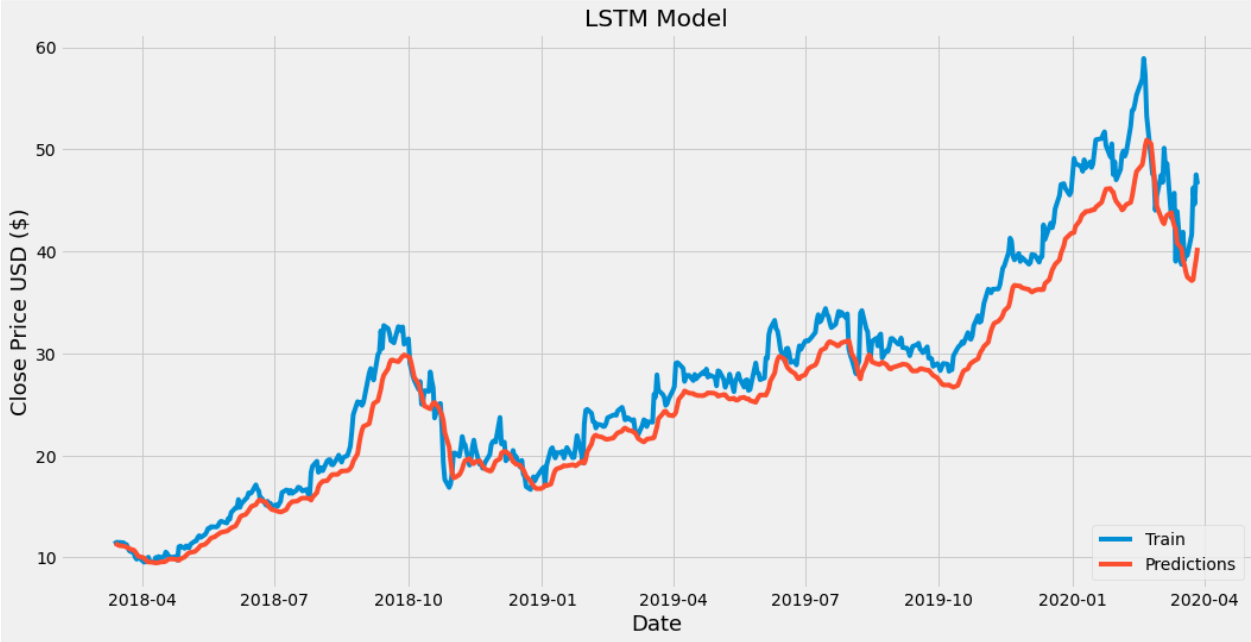

Figure 28 – Plotting the predictions

Now the predictions can be visualized on a line graph. When observing figure 28, the model was somewhat accurate in its predictions of how the stock price would behave, however, it generally undervalued how the stock would perform. The model becomes increasingly less accurate as the 2020 stock data is shown. This is most likely due to the erratic nature of the stock price caused by outside variables that the model cannot factor.

Date	Close	Predictions
2018-03-13	11.640000	11.377663
2018-03-14	11.360000	11.312766
2018-03-15	11.460000	11.220824
2018-03-16	11.470000	11.151944
2018-03-19	11.430000	11.105664
...	...	...
2020-03-23	41.639999	37.117245
2020-03-24	46.220001	37.238155
2020-03-25	44.630001	38.301003
2020-03-26	47.500000	39.177406
2020-03-27	46.580002	40.339222

Figure 29 – point-by-point data

Figure 29 shows a closer look at the values that the model predicted for the first five and last five data points. At the start of the predictions, the model was never more than fifty cents off with its predictions, however towards the end of the predictions, the model in some predictions is close to ten dollars off with its predictions.

```
percent_difference = abs(valid['Predictions'] - valid['Close']) / ((valid['Predictions'] + valid['Close']) / 2) * 100


percent_difference
```

```
Date
2018-03-13     2.279447
2018-03-14     0.416655
2018-03-15     2.109057
2018-03-16     2.811925
2018-03-19     2.878425
                 ...
2020-03-23    11.485305
2020-03-24    21.524189
2020-03-25    15.263287
2020-03-26    19.203606
2020-03-27    14.359953
Length: 515, dtype: float64
```

Figure 30 – percent difference

Figure 30 shows the percent difference between the predictions and the actual closing value of the stock price. As observed in figure 29, the predictions are close at the beginning of the simulation but become less accurate as the simulation reaches the end.

```
amd_quote = web.DataReader('AMD', data_source='yahoo', start='2010-01-01', end='2020-3-27')
new_df = amd_quote.filter(['Close'])
last_60_days = new_df[-60:].values
last_60_days_scaled = scaler.transform(last_60_days)
X_test = []
X_test.append(last_60_days_scaled)
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
pred_price = model.predict(X_test)
pred_price = scaler.inverse_transform(pred_price)
print(pred_price)
```

```
[[41.183624]]
```

```
amd_quote2 = web.DataReader('AMD', data_source='yahoo', start='2020-03-30', end='2020-3-30')
print(amd_quote2['Close'])
```

```
Date
2020-03-30    47.860001
2020-03-30    47.860001
Name: Close, dtype: float64
```

```
percent_difference2 = abs(41.183624 - 47.860001) / ((41.183624 + 47.860001) / 2) * 100

percent_difference2
```

```
14.99574394011923
```

Figure 31 – One day prediction

Figure 31 is a miniature version of what the model performed with the original training

data. However, in figure 31 the model is attempting to predict the closing price for March 30th, 2020 which is a date that is not included in the original training data. The model predicted a closing price for March 30th to be 41.183624 while the actual closing price for March 30th was 47.860001. This prediction has a percent difference of roughly fifteen percent.

Conclusion

Monte Carlo simulations and LSTM models provide some interesting observations when looking at stock data. With the Monte Carlo simulation in figure 16, the end points for prices tend to be clustered together, however the way that the iterations in the Monte Carlo simulation arrive at these end points have some differences. An analyst observing the output in figure 16 may conclude that there is a likely chance the stock price will increase to around eighty dollars over the next 365 days. While the output provided by the simulation has some interesting trends, it is unknown if these outputs will be completely accurate with the real-world stock data.

The LSTM model provided some interesting predictions as well. The model began with predicting AMD stock rather well with only a two percent difference between the actual closing price of AMD stock and the predicted closing price. As the model progressed over time it became less accurate with the final five data points having as high as a twenty-one percent difference. The degradation of the model's performance could be influenced by the volatility of stock market prices. This model is using closing prices during the COVID-19 pandemic as training data, which has notably caused the stock market to be in flux. While the model's point-by-point performance degraded over time, when observing the model on a graph, like in figure 28, the model managed to predict the general trend of how the stock price would behave. The model consistently undervalued the stock price in its predictions, but most of the growth and loss of the stock price was predicted at the correct time on the model.

There are ways that the LSTM model could be further tuned to provide more accurate predictions. One way would be to tune the batch size and number of epochs that the model runs. In the model that was demonstrated only one epoch was performed, therefore there was only one iteration of the simulation that occurred. Due to the processing power and time required to perform these calculations, more epochs were not performed on this model. However, if the model attempted multiple iterations, it could come to a more accurate conclusion.

Research has been performed on combining the LSTM model with other types of machine learning or regressive models to provide more accurate predictions. Examples include, combining LSTM with a point swarm optimization algorithm (PSO) to predict the price of nickel (Shao, Li, Zhao, Bian, 2019) and combining LSTM with the auto-regressive integrated moving average (ARIMA) to predict housing sales (Ayse, Akgun, Temur, 2019).

While Monte Carlo simulations and LSTM models may not find use among daily stock traders, these two models can be useful for business that wish to see predictions on how their stock price may perform over time. These models are never completely accurate and due to the variability of the stock market they will likely never be completely accurate with their predictions. However, both simulations are adept at predicting how a stock price behaves in a general sense, and not as a point-by-point analysis.

References

Ayşe Soy Temür, Akgün, M., & Günay Temür. (2019). Predicting housing sales in turkey using ARIMA, LSTM and hybrid models. *Journal of Business Economics and Management, 20*(5), 920-938. doi:http://dx.doi.org/10.3846/jbem.2019.10190

Balenthiran, K. (2013). *17.6 year stock market cycle : Connecting the panics of 1929, 1987, 2000 and 2007*.

Brownlee, J. (2019, August 14). A Gentle Introduction to Exploding Gradients in Neural Networks. Retrieved March 13, 2020, from https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

Brownlee, J. (2017, June 7). The 5 Step Life-Cycle for Long Short-Term Memory Models in Keras. Retrieved March 31, 2020, from https://machinelearningmastery.com/5-step-life-cycle-long-short-term-memory-models-keras/

Olah, C. (2015, August 27). Understanding LSTM Networks. Retrieved March 13, 2020, from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow concepts, tools, and techniques to build intelligent systems*. Beijing: OReilly Media.

Guttag, J. (2014). *Introduction to Computation and Programming using Python* (2nd ed.). Cambridge, Mass: The MIT Press.

Hardesty, L. (2017, April 14). Explained: Neural networks. Retrieved March 13, 2020, from http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

Jain, V. R., Gupta, M., & Singh, R. M. (2018). Analysis and prediction of individual stock prices of financial sector companies in NIFTY50. *International Journal of Information Engineering and Electronic Business, 10*(2), 33. doi:http://dx.doi.org/10.5815/ijieeb.2018.02.05

Krizanová, A., Masárová, G., Majercák, P., & Buc, D. (2013). Monte carlo cost simulation in the supply chain in E-business/Monte carlo simulacija troskova u opskrbnom lancu kod e-poslovanja. *Nase More, 60*(5), 99-104.

Moody, Michael J,M.B.A., A.R.M. (2013). PREDICTIVE MODELING: BETTER DECISION MAKING. *Rough Notes, 156*(4), 42-42,44

Müller Andreas Christian, & Guido, S. (2018). *Introduction to machine learning with Python a guide for data scientists*. Beijing: OReilly.

Namin, S., Namin, A. (2018). Forecasting Economic and Financial Time Series: ARIMA vs. LSTM. [PDF file]. Lubbock, TX.

NIST. (n.d.). Related Distributions. Retrieved March 31, 2020, from https://www.itl.nist.gov/div898/handbook/eda/section3/eda362.htm

Palisade. (n.d.). What is Monte Carlo Simulation? Retrieved March 13, 2020, from https://www.palisade.com/risk/monte_carlo_simulation.asp#:~:text=

Predictive Success Corporation (2019, May 6). A Brief History of Predictive Analytics. Retrieved March 13, 2020, from https://medium.com/@predictivesuccess/a-brief-history-of-predictive-analytics-f05a9e55145f#:~:text=

Provost, F., & Fawcett, T. (2013). *Data science for business: what you need to know about data mining and data-analytic thinking*. Sebastopol: OReilly.

SAS. (n.d.). Artificial intelligence, machine learning, deep learning and more. Retrieved March 13, 2020, from https://www.sas.com/en_us/insights/articles/big-data/artificial-intelligence-machine-learning-deep-learning-and-beyond.html#:~:text=

SAS. (n.d.). Predictive Analytics: What it is and why it matters. Retrieved from https://www.sas.com/en_us/insights/analytics/predictive-analytics.html#dmtechnical

Shao, B., Li, M., Zhao, Y., & Bian, G. (2019). Nickel price forecast based on the LSTM neural network optimized by the improved PSO algorithm. *Mathematical Problems in Engineering*, *2019*, 15. doi:http://dx.doi.org/10.1155/2019/1934796

Appendix A

Python Libraries	
Library Name	**Library Function**
datetime	Supplies classes for manipulating dates and times
math	Provides access to various mathematical functions defined by the C standard
matplotlib	Primary scientific plotting library. Provides functions for visualizations such as line charts, histograms, etc.
NumPy	Contain functionality for multidimensional arrays and other high-level math functions such as linear algebra and pseudorandom number generators.
pandas	Library built for data analysis. Uses a structure called a DataFrame which, unlike NumPy arrays, allows each column of data to have different data types. Pandas also grants Python the ability to read external data files such as Excel spreadsheets and CSV files.
pandas_datareader	This is used to create pandas DataFrames from different sources via the internet such as Yahoo! Finance.
scikit-learn	A free machine learning library that provides access to various machine learning algorithms.
TensorFlow	Python library created by Google that allows for fast numerical computing and the creation of Deep Learning models.

Appendix B

Monte Carlo Simulation code

```
import numpy as np
import pandas as pd
from pandas_datareader import data as wb
import matplotlib.pyplot as plt
from scipy.stats import norm
# %matplotlib inline

ticker = 'AMD'
data = pd.DataFrame()
data[ticker] = wb.DataReader(ticker, data_source='yahoo', start =
'2010-1-1')['Adj Close']

log_returns = np.log(1 + data.pct_change())
log_returns.tail()

u = log_returns.mean()
u

var = log_returns.var()
var

drift = u - (0.5 * var)
drift

stdev = log_returns.std()
stdev
```

```
t_intervals = 365
iterations = 10

daily_returns = np.exp(drift.values + stdev.values *
norm.ppf(np.random.rand(t_intervals, iterations)))

daily_returns

S0 = data.iloc[-1]
S0

price_list = np.zeros_like(daily_returns)
price_list

price_list[0] = S0
price_list

for t in range(1, t_intervals):
  price_list[t] = price_list[t - 1] * daily_returns[t]
price_list

plt.figure(figsize = (10, 6))
plt.plot(price_list)
plt.title('Monte Carlo Simulation (AMD)')
plt.xlabel('Num. of Days')
plt.ylabel('Price USD ($)')
```

Appendix C

LSTM Code

```
import math
import pandas_datareader as web
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, LSTM
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

df = web.DataReader('AMD', data_source='yahoo', start='2010-01-01',
end='2020-4-6')
df

data = df.filter(['Close'])
dataset = data.values
training_data_len = math.ceil( len(dataset) * .8 )

training_data_len

scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)
scaled_data

train_data = scaled_data[0:training_data_len , :]
```

```
x_train = []
y_train = []

for i in range(60, len(train_data)):
  x_train.append(train_data[i-60:i, 0])
  y_train.append(train_data[i,0])
  if i<= 61:
    print(x_train)
    print(y_train)
    print()

x_train, y_train = np.array(x_train), np.array(y_train)

x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
x_train.shape

model = Sequential()
model.add(LSTM(50, return_sequences =True, input_shape =
(x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences = False))
model.add(Dense(25))
model.add(Dense(1))

model.compile(optimizer = 'adam', loss = 'mean_squared_error')

model.fit(x_train, y_train, batch_size=1, epochs=1)

test_data = scaled_data[training_data_len - 60:, :]
```

```
x_test = []
y_test = dataset[training_data_len:, :]
for i in range(60, len(test_data)):
  x_test.append(test_data[i-60:i, 0])

x_test = np.array(x_test)

x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)

rmse = np.sqrt(np.mean(((predictions - y_test)**2)))
rmse

train = data[:training_data_len]
valid = data[training_data_len:]
valid['Predictions'] = predictions
plt.figure(figsize=(16, 8))
plt.title('LSTM Model')
plt.xlabel('Date', fontsize=18)
plt.ylabel('Close Price USD ($)', fontsize=18)
plt.plot(valid[['Close', 'Predictions']])
plt.legend(['Train', 'Predictions'], loc = 'lower right')
plt.show()

valid
```

```
percent_difference = abs(valid['Predictions'] - valid['Close']) / ((valid['Predictions'] + valid['Close']) / 2) * 100

percent_difference


amd_quote = web.DataReader('AMD', data_source='yahoo', start='2010-01-01', end='2020-4-6')

new_df = amd_quote.filter(['Close'])

last_60_days = new_df[-60:].values

last_60_days_scaled = scaler.transform(last_60_days)

X_test = []

X_test.append(last_60_days_scaled)

X_test = np.array(X_test)

X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

pred_price = model.predict(X_test)

pred_price = scaler.inverse_transform(pred_price)

print(pred_price)


amd_quote2 = web.DataReader('AMD', data_source='yahoo', start='2020-04-7', end='2020-4-7')

print(amd_quote2['Close'])


percent_difference2 = abs(44.435703 - 47.560001) / ((44.435703 + 47.560001) / 2) * 100


percent_difference2
```